Polynomial Regression on Finance Trends

Francesco Romito, Marco Aspromonte

Abstract

In the following report we will discuss about the details of implementation of the Polynomial Regression for Finance Trends about short term predictions, in order to evaluate how the model is able to estimate the real price trends of an asset. The model is written in a Colab notebook. For testing reasons and also for personal interests, we evaluated a Bitcoin's trend period. However the reliability of the model is more convenient much more for short periods.

1 The Dataset

The Bitcoin trend dataset has been downloaded from `Yahoo Finance!` from 2014/09/17 to 2021/05/19. It is composed by 7 main attributes. For the model we do not consider the column 'Adj Close' since it is very similar to the 'Close' one. Here we have a synthetic representation of the first 5 elements of the dataframe. All the elements are floating point numbers, except for 'Date' that is a String, so it is necessary to encode it in a appropriate numeric form.

	Date	Open	High	Low	Close	Adj Close	Volume
0	2019-09-18	9253.401367	9316.917969	9240.340820	9246.486328	9246.486328	1.466026e+10
1	2019-09-19	9248.524414	9319.454102	8917.574219	9292.973633	9292.973633	1.804724e+10
2	2019-09-20	9292.886719	9334.069336	9194.604492	9239.483398	9239.483398	1.337076e+10
3	2019-09-21	9241.304688	9245.341797	9075.292969	9092.541992	9092.541992	1.218296e+10
4	2019-09-22	9096.534180	9142.628906	9004.768555	9138.951172	9138.951172	1.197878e+10

Figure 1: Dataset

2 Preprocessing

Since we have to work with a large and various dataset, it was necessary to implement a cleaning and normalization process. So, after the import of the `csv` dataset and the conversion in a `pandas` dataframe, since Volume column is strongly variable, first we check for its outliers that could effect negatively the model, so we remove each row that contains one of them. A problem occured in the management of the Date column is that it was defined as a String type, so it was convenient to encode it in an appropriate form, i.e. `datetime64[ns]` , then map it into an `int64` trough the function `map(dt.datetime.toordinal)`. After a drop process for NaN values, in order to generalize and make more robust the model with a normal distribution, we have applied the Standard Score normalization to the dataframe :

$$\frac{(X-\mu)}{\sigma}$$

For the following steps, the function returns the cleaned dataframe, the encoded Date column `X` ,the `ratio` and the original minimum value.

```
def prepare_dataset(df):
  df=df[~(np.abs(df.Volume-df.Volume.mean()) > (3*df.Volume.std()))]
  df['Date'] = pd.to_datetime(df['Date'])
  df['Date'] = df['Date'].map(dt.datetime.toordinal)
  df = df.dropna(axis=0)
```

```
  dist = df.max()-df.min()
  original_min = df.min()
  std_dev = df.std()
  mean = df.mean()
  df = (df - df.mean())/df.std()
  X = df['Date']
  ratio = dist/(df.max()-df.min())
return df, X, ratio, original_min
```

3 Make a prediction

3.1 Linear Regression with Gradient Descent

The idea is that some of the values have a linear distribution, so we first define the relationships btw the Open values (that is the dependent variable `Y`) and all the other independent ones `X` trough a linear regression, minimizing the following mean square error:

$$\frac{1}{n}\sum_{i}^{n}(y_i - (y_i^p))^2$$

The main function of prediction is called `predict_n_day_after` that is formally the main execution flow.

```
def predict_n_day_after(df, n, stamp = True):
  df_base = df.copy()
  df, X, ratio, original_min = prepare_dataset(df)
  Y = df.iloc[:, 1]
  L = 0.1
  ...
  epochs = 100
  m_high, c_high = plot_regression(df.iloc[:, 2], Y, L, epochs, 'High values',
                                   'Open values', stamp = stamp)
......
```

So after that the dataset is prepared and the execution parameters are tuned (learning rate, epochs), we can call the `plot_regression` function with such parameters. So here, in order to minimize the error we perform a simple Gradient Descent algorithm, because now it is still sufficient for this purpose:

```
def plot_regression(X , Y, L, epochs, x_argument, y_argument, stamp = True):
    m = 0
    c = 0
    for i in X:
        n = float(len(X))
        Y_pred = m*X + c
        D_m = (1/n) * sum(X * (Y - Y_pred))
        D_c = (1/n) * sum(Y - Y_pred)
        m = m - L * D_m
        c = c - L * D_c
        ...
    return(m,c)
```

Starting with an initialization of the weights with zero, we define the derivative of the error function wrt them, and finally they are updated by subtracting the current value with the derivative multiplied by the learning rate. Once the coefficient are returned, then we can use them to make the prediction, so to define the regression lines between Open and the other independent variables. Here there are some graphic results of the above mentioned lines.

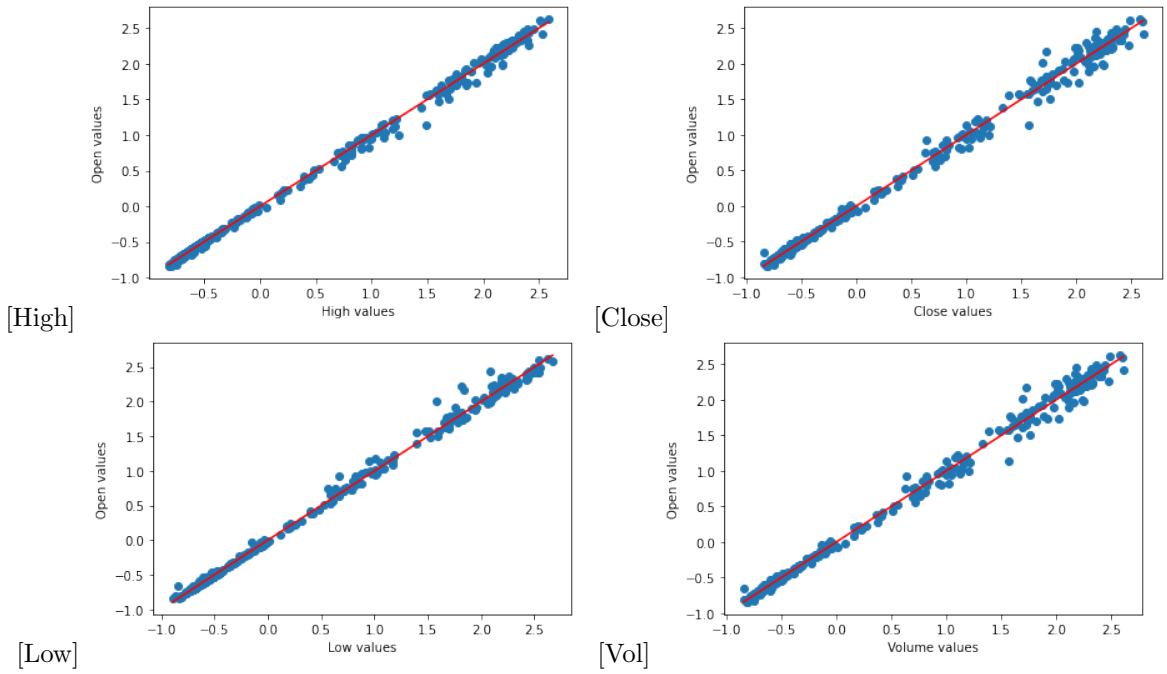

Figure 2: Relationships between Open and the other values

3.2 Polynomial regression with Batch Stochastic Gradient Descent

The next step concerns to find the curve that represents the time series of the opening values. To do this we need a polynomial degree at least of the third order, and we need to tune more then the two previous coefficients, so a simple GD could not be enough. A powerful method to deploy this consists in consider a polynomial function of 5th degree. So we tune the six weights deploying the Batch Stochastic Gradient Descent. They are formally performed trough the function `stochastic_gradient_descent` that belongs in `plot_mult_regression`, and returned by this last mentioned function:

```
def plot_mult_regression(df,Y, L, epochs, decay_rate, batch_size, stamp=True):
  X = df.iloc[:, 0]
  n_vars=6
  r = stochastic_gradient_descent(X, Y, n_vars, learn_rate=L, decay_rate =decay_rate,
                                  batch_size = batch_size, n_iter = epochs)
  Y_pred = r[5]*X**5 + r[4]*X**4 + r[3]*X**3 + r[2]*X**2 + r[1]*X + r[0]
  ...
  return(r[1],r[2],r[3],r[4],r[5],r[0])
```

Then we perform the Batch Stochastic Gradient Descent function:

```
def stochastic_gradient_descent(x, y, n_vars=None, start=None, learn_rate=0.1,
    decay_rate=0.0, batch_size=1, n_iter=50, tolerance=2*10^-6):
    ...
    for _ in range(n_iter):
        rng.shuffle(xy)
        for start in range(0, n_ele, batch_size):
            end = start + batch_size
            x_batch, y_batch = xy[start:end, :-1], xy[start:end, -1:]
            grad = np.array(gradient_6_vars(x_batch, y_batch, weights), np.dtype("float64"))
            difference = decay_rate * difference - learn_rate * grad
            if np.all(np.abs(difference) <= tolerance):
                break
            weights += difference
    return weights
```

The previous function also calls `gradient_6_vars`,which computes the update of the weights based on the sum of square error:

```
def stochastic_gradient_descent(x, y, n_vars=None, start=None, learn_rate=0.1,
def gradient_6_vars(x, y, w):
    res = w[0] + w[1] * x + w[2] * x**2 + ... + w[6]* x**5 -y
    return res.mean(), (res * x).mean(), (res * x**2).mean()...
```

The main difference with the previous implementation of the GD is that now the gradient is evaluated by considering the whole dataset divided in minibatches which contains random picked samples, where the size of the minibatches is assigned by the `batch_size` variable. This of course is more efficient in terms of computational effort, and since we have a large dataset is a good behaviour to reduce the cost. Moreover in the update process of the weights, besides considering only the learning rate, a momentum defined trough the `decay_rate` is set to help the algorithm to converge easier. If the weight's updates are close to zero, the assignment is break and they are returned by the function. This is done by a tuned value of tolerance. Now we have all the elements to show the results of the Polynomial Regression:

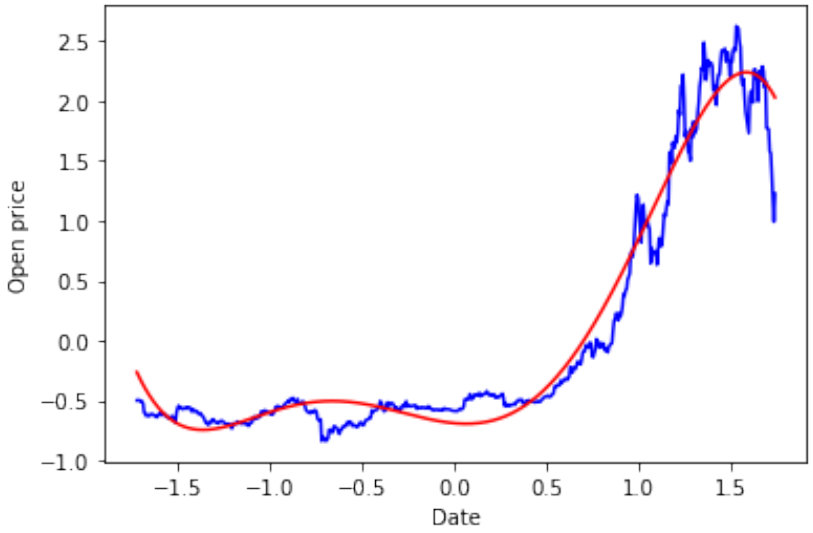

Figure 3: Polynomial function and the real open price

3.3 Prediction

Now that we have obtained all the necessary weigths, we can predict a new opening value given a number of days after the last recorded value. To do this we choose an appropriate definition of X(`X_pred`) which represents the day we want to forecast, and the output (`y_pred`) of the polynomial function will be the predicted-normalized value of Open, as follows:

```
....
unit = 2*X.max()/X.shape[0]
X_pred =  X.max() + n* unit
y_open = (op5*X_pred**5 + op4*X_pred**4 + op3*X_pred**3 + op2*X_pred**2 + op1*X_pred + bop)
....
```

Given this value we can now forecast also the other ones through the linear mapping that we have defined previously with the Gradient Descent:

```
...
y_high_pred = (y_open - c_high)/ m_high
y_low_pred = (y_open - c_low )/m_low
y_close_pred = (y_open - c_close)/ m_close
```

```
y_volume_pred = (y_open - c_volume) / m_volume
...
```

3.3.1 Denormalization

Finally, in order to define real prices numbers we have to denormalize the predicted values, so here we go with the `denormalization` function, which add to the normalized value the minimum normalized one. Then it is multiplied by the ratio of its own class and at the end the real minimum is added, because the real minimum of the Bitcoin's price in our dataset is of course ever greater than zero.

```
def denormalize(df, ratio, original_min, y, index):
    y = (abs(y) + abs(df.iloc[:, index].min()))*ratio[index] + original_min[index]
    return y
...
y_open = denormalize(df, ratio, original_min, y_open, 1)
...
```

3.3.2 Some results

Here we can show some of the prediction's results, with `n_days_after` set to 1, so the predicted value is for the successive day wrt the last day of the dataset:

- The opening predicted value is : 44546
- The high predicted value is : 45924
- The low predicted value is : 43092
- The close predicted value is : 44693
- The volume predicted value is : 683182166340992

In general we can consider that these values are acceptable wrt the real values that usually are sensitive to an undefined number of external variables that we can't really consider in this model, except for Volume since it is too much unpredictable because it is not strictly related to the Opening value, so it's prediction is not amenable by exploiting Opening prices.

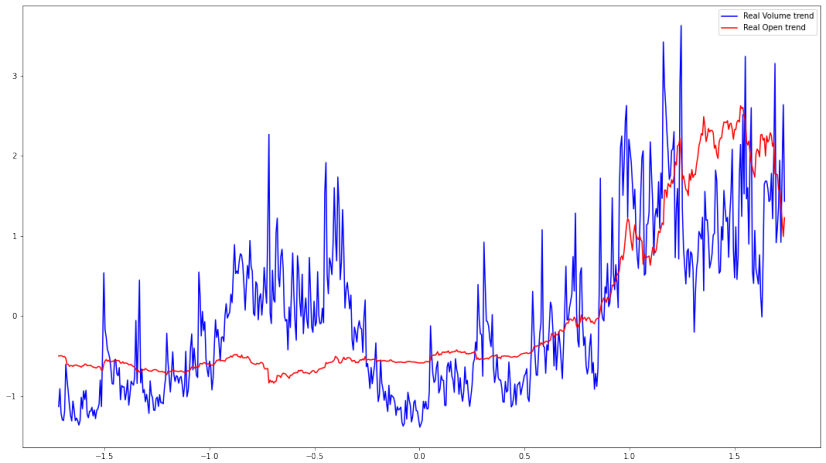

Figure 4: Real trend of Open and Volume

4 Evaluation of the model

4.1 Rolling Window

In order to test a more detailed reliability of the model it is useful to evaluate the mean error with a rolling window, that applies the model on different slices of 30 days (defined in `n_slice`), wrt the days we have tuned in the previous section. This is made by the `rolling_window` function that, after computing each prediction value, computes a mean relative error vector between the predicted values and real ones.

```
def rolling_window(df, n_days_after, n_slice):
  for i in range(0,n_slice):
    df_month = df.iloc[30*i:30+30*i].copy()
    true_values = df.iloc[30+30*i+n_days_after].copy()
    y_open, y_high_pred, y_low_pred, y_close_pred, y_volume_pred =/
            predict_n_day_after(df_month, n_days_after, stamp=False)

    perc_err_open = abs(y_open - true_values['Open'])/true_values['Open']
    perc_err_high = abs(true_values['High']-y_high_pred)/true_values['High']
    ...
    loss.append([perc_err_open, perc_err_high, perc_err_low, perc_err_close, perc_err_volume])

  mean_loss_open, mean_loss_high, mean_loss_low,mean_loss_close, mean_loss_volume =/
                                                                  0, 0, 0, 0, 0

  for i in range(0, len(loss)):
    mean_loss_open = mean_loss_open + loss[i][0]
    mean_loss_high = mean_loss_high + loss[i][1]
    ...

  mean_loss_open = round(mean_loss_open / len(loss)*100)
  mean_loss_high = round(mean_loss_high / len(loss)*100)
  ...
```

So we can see the error's percentage of the polynomial regression curve with respect to the entire dataset in slices of 30 days:

- the mean of the error for open is: 14.7 %
- the mean of the error for high is: 14.2 %
- the mean of the error for low is: 13.0 %
- the mean of the error for close is: 14.1 %
- the mean of the error for volume is: 150112.6 %

In according to the previous considerations about the reliability of the values, also from the error we can see how the values are acceptable (except for Volume of course), but another limit is given by the idea that the structure of the function used in Open's prediction is designed in principle for the whole dataset, and also it needs another fitted tuning procedure of the parameters for each slice(e.g. `learning_rate` etc..).

4.2 Comparison with SkLearn's SVRs

To do an extra final comparison of the model in order to verify the accuracy, we also plot a comparison with some SVRs, one of `SVR rbf kernel` and the other one with the `polynomial kernel` noticing that our model is extremely better than the first and pretty similar to the second one. Here we have a graphical representation:

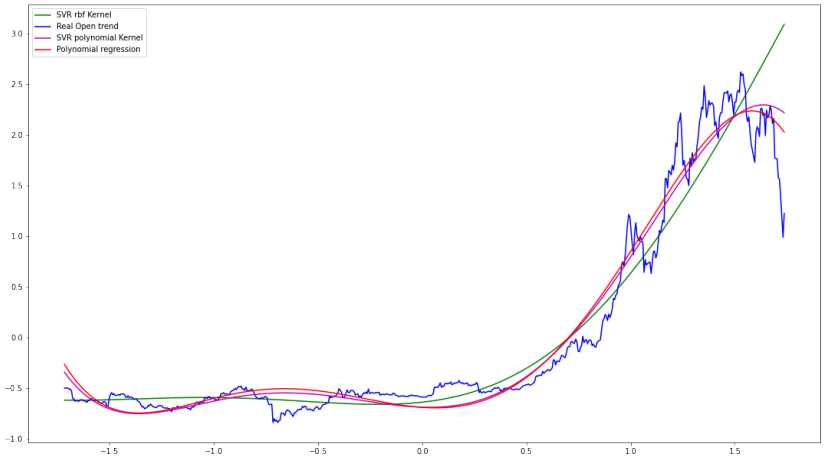

Figure 5: SkLearn comparisons

5 Conclusion

To conclude, we can say that GD is an effective technique in order to tune parameters in a linear regression, while when we have more weights to calculate it is better to use a more complex method as the BSGD. Overall the methods fits pretty the curve but of course, due to more general influences that characterize the uncertainty on finance trends, in particular for long terms, it could not be use as a working model for own investments but it must be seen more as an advice, especially for very short terms.

References

[Vittorio Maniezzo] Combinatorial Decision Making and Optimization's slides

[OpenCV] Documentation on https://pandas.org

[Numpy] Documentation on https://numpy.org

[Numpy] Documentation on https://sklearn.org